To Appear in the Proceedings of 12th International Conference on Parallel Architecture and Compilation Techniques (PACT), 2003.

Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU

Mauricio Breternitz, Jr.
Microprocessor Research Labs,
Intel Corporation
mauricio.breternitz.jr@intel.com

Herbert Hum
Desktop Platforms Group,
Intel Corporation
herbert.hum@intel.com

Sanjeev Kumar
Microprocessor Research Labs,
Intel Corporation
sanjeev.kumar@intel.com

Abstract

Graphics and media processing is quickly emerging to become one of the key computing workloads. Programmable graphics processors give designers extra flexibility by running a small program for each fragment in the graphics pipeline. This paper investigates low-cost mechanisms to obtain good performance for modern graphics programs on a general purpose CPU.

This paper presents a compiler that compiles SIMD graphics program and generates efficient code on a general purpose CPU. The generated code can process between 25–0.3 million vertices per second on a 2.2 GHz Intel Pentium® 4 processor for a group of typical graphics programs.

This paper also evaluates the impact of three changes in the architecture and compiler. Adding support for new specialized instructions improves the performance of the programs by 27.4 %. on average. A novel compiler optimization called mask analysis improves the performance of the programs by 19.5 % on average. Increasing the number of architectural SIMD registers from 8 to 16 registers significantly reduces the number of memory accesses due to register spills.

1 Introduction

Graphics and media processing is quickly emerging to become one of the key computing workloads [3, 16]. The need for realism in interactive 3D Games [8] and techniques such as ray-tracing are some of the reasons for increased computational demands [20]. System and processor designers address this computation workload via media extensions to instruction sets [14, 19, 15] and specialized graphics coprocessors [2]. More recently *programmable* graphics coprocessors [10] are intended to replace the fixed-function graphics pipeline.

Programmable graphics processors give 3D graphics designers the flexibility of running a small program for each vertex in the graphics pipeline. These small programs, called *vertex shaders*, enable 3D graphics designers, such as game developers, to achieve tight control of operations in the graphics pipeline and provide more realistic effects. In the absence of high-end graphics cards and coprocessing accelerators, these vertex shaders are run on the host processor.[1]

The goal of this project is to obtain good performance for graphics programs without the cost of graphics hardware. The approach is to investigate new compiler optimizations along with extensions to the instruction set for execution of vertex shaders by a general purpose CPU.

This paper examines possible extensions to a media instruction set. It uses the SSE2 [17] instruction set as a case study. This paper first presents an analysis of the performance of shaders on the base instruction set, and then discusses the relative impact of the architectural and compiler enhancements.

A retargetable optimizing compiler has been implemented. It accepts a stylized description of new instructions and generates efficient code for a set of vertex shaders. A new compiler optimization that is well suited for the operations found in graphics programs has been developed and implemented.

Based on our analysis, we describe the impact of three changes in the architecture and compiler: adding support for new instructions such as dot-product; increasing the number of SIMD registers provided by the architecture; and implementing a novel compiler optimization. The combined improvements obtain about twice the performance at a limited hardware cost.

[1] In some cases, even when some graphics support is available (e.g. Intel 845G integrated graphics chipset), vertex shaders are run on the host processor.

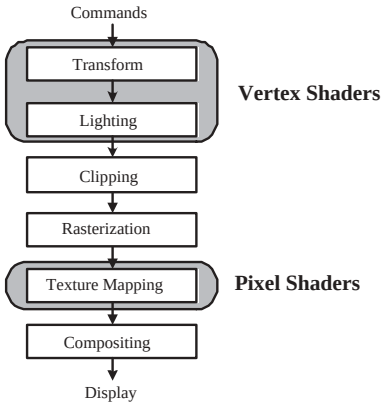

Figure 1. Graphics pipeline

2 Background

This section starts with a detailed description of the graphics pipeline and vertex shader programs. It also provides a brief description of the SIMD support available on Intel Pentium® processors.

2.1 Programmable Graphics Pipeline

Graphics processing has traditionally been accelerated with a highly tuned fixed-function pipeline (Figure 1) which is implemented in hardware. The pipeline takes "commands" (shapes like triangles that describe the scene being rendered) and outputs a two dimensional image that can be displayed on the screen. The operations in the graphics pipeline are divided into a number of major categories: transform, lighting, clipping, rasterization, texture mapping, and compositing.

Hardware implementations of the graphics pipeline implement only a limited selection of graphics effects. To alleviate this, the new generation of graphics cards [10] allow portions of the graphics pipeline to be programmed by the graphics applications' developer. On these programmable graphics cards, the developer can specify the transform and lighting phases of the pipeline by writing *vertex shader* programs. The developer can also program the texture mapping phase by writing *pixel shader* (also called *fragment shader*) programs. Due to processing requirements and resource constraints in the graphics engine, these are small assembly-level programs.

Vertex Shader. A vertex shader operates on a vertex at a time as input, processes it, and generates the output vertex. No global state is maintained across vertices.

3D Graphics objects are described according to different frames of reference at several steps during processing. *World space* coordinates hold all 3D objects that are part of the 3D world. *Eye space coordinates* are used for lighting and culling. Screen space coordinates are used to store the scene in the graphics frame buffer. The **transform** step converts element coordinates from one frame of reference to another, e.g., between world space and eye space.

Lighting effects are essential for obtaining realism in a scene. Triangle setup/clipping prepares the triangle data for the rendering engine. Rendering computes the correct color of each pixel on the screen, using data from the setup stage. The rendering engine considers the color of the object, color and direction of light incident on the object, and texture properties of the object. 3D **Lighting** attempts to model the physics of light reflection on the object. To simplify processing it models the light reflection in two major components: diffuse and specular reflection. Diffuse reflection assumes that light hitting an object scatters in all directions. Specular reflection depends on the position of the viewer as well as the direction of light and orientation of the triangle being rendered. It captures the reflection (mirror-like) properties of the object to achieve reflection and glare effects.

Pixel Shader. The pixel shader runs after the rasterization[2] phase and processes a pixel at a time. It implements a number of operations such as texturing, filtering, blending, fog blending, and visibility testing.

2.2 Shaders on General-Purpose Processors

In the absence of high-end graphics cards and coprocessing accelerators, the vertex shaders can be run on the host processor and achieve reasonable performance. [3] However, this continues to be a challenging problem for several reasons. First, when a vertex shader is run on the host CPU, it gets only a fraction of the processing time because it has to share it with the graphics application as well as the operating system. Second, graphics processors are getting faster at a faster rate than general-purpose processors.

Unlike vertex shaders, pixel shaders are rarely run on the host processor. During rasterization, each object (like a triangle) yields a number of pixels. So the pixel shader has an order of magnitude more inputs to process than the vertex shader. This causes two problems. First, the computational requirements of pixel shaders are much higher than vertex shaders. This is the reason why only a small fraction of computational resources on high-end graphics cards is devoted to vertex shader processing. Second, the band-

[2]The rasterization phase converts the various objects (like triangles) into *Pixels*.

[3]For instance, the performance of typical vertex shaders on a general-purpose processor (2.8 GHz Intel Pentium 4 processor) is about 40 % of the performance on a high-end graphics card (ATI Rad9700) [21]. These numbers were obtained from Intel's production shader compiler.

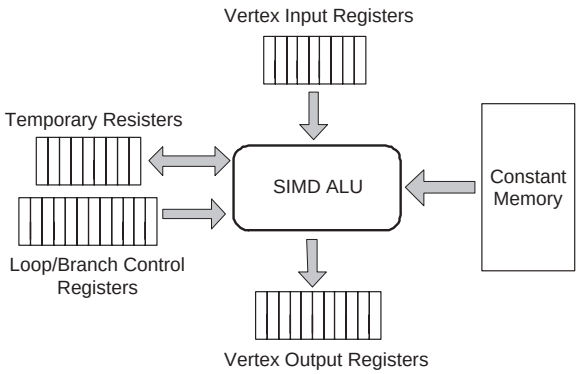

Figure 2. Vertex Shader Virtual Machine

width requirement for pixel shaders is significantly higher than what is available to the host processor.

Desktop graphics applications are designed to adapt to the amount of graphics support available on the machine. On machines that have high-end graphics cards, these applications would use vertex and pixel shaders to render more realistic scenes. On machines that do not have special-purpose hardware to run the shaders, they can use the host processor to run just the vertex shaders and produce reasonable scenes. This paper focuses on running vertex shaders efficiently on the host processor.

2.3 Vertex Shader Virtual Machine

The virtual machine (Figure 2) executing vertex programs has input ports for incoming vertex data, a small array to hold program constant data, a number of registers for temporary variables and output ports. Graphics data is expressed as a vector with vertex homogeneous position (x, y, z, and w coordinates), along with other vertex data such as fog and diffuse and specular colors.

Microsoft has standardized the requirements on the vertex and pixel shaders that are supported in DirectX [12]. For example, for vertex shaders the virtual machine must support the following minimum amount of registers:

- 16 vertex input registers
- 96 constant registers (not changed by the shader)
- 12 temporary registers
- The sequence of commands can be up to 128 operations long.

The shader virtual machine instruction set has a number of SIMD-like operations such as ADD, MUL and 3-element and 4-element dot product. Data coordinates are expressed in fixed-point and low-precision floating point formats. The set of commands which the virtual machine must execute is fairly large and includes:

```
dp4   oPos.x, v0, c[0]
dp4   oPos.y, v0, c[1]
dp4   oPos.z, v0, c[2]
dp4   oPos.w, v0, c[3]

mov   oD0,  c[4]
```

Figure 3. A Simple Vertex Shader

- addition, subtraction, multiplication of vectors by a number
- dot product and cross product of vectors
- multiplication with addition
- sampling of minimum and maximum values; setting the register at 0 or 1 if the value is smaller or greater than the said one
- exponent and logarithm
- macro-operations of multiplying the vector by matrix of different sizes

Figure 3 illustrates a simple shader. This shader applies a constant color to each vertex. The first four `dp4` instructions are a matrix multiply to transform the object vertices with a view/projection matrix. The `dp4` operator designates a four-component dot-product (each of the instruction arguments is a vector with single-precision floating-point values). The vertex output register is designated 'oPos' and the mask (operator '.') designates the x, y, z or w component of the result. The final instruction, `mov`, copies the constant color value to the output register that holds the diffuse color component of the vertex. The `mask` operator selects which sub-fields of the vector are affected by the result of the operation.

Vertex shader programs frequently use 'mask' and 'swizzle' attributes to their instructions. *Mask* selects components of a vertex that are not affected by the operation. It can select individual fields (as in the above example) and may also specify replication of a given field in the final vector. Mask fields are frequently used to operate on the x, y and z coordinates only. *Swizzle* reorders and/or replicates components of the vector operands to the instruction.

For example, one may use a mask to add only the x and y components of a vector, and not affect the z and w fields. Also, we specify a swizzle factor to 'reorder' the second operand to, say, add the x, y, z, w components of a vector respectively to the w, z, y, x components, returning the result in the x, y, z, w slots.

2.4 SIMD support on Pentium® Processors

The SSE2 [17] instruction set supports SIMD operations on four-wide single-precision floating-point vectors. It contains 8 architectural registers and provides a comprehensive

set of basic arithmetic operations such as addition and multiplication. It also supports efficient 128-bit bitwise logical operations on register contents which can be used to implement element-wise masking required by the shader instruction set. The programmer can specify cacheability hints for efficient processing of streaming data.

The SSE2 instruction set is a good match for vertex shader code generation because of the 128-bit vector size and the register file. Many basic shader operations may be translated into a single SSE2 instruction. Temporary values are stored in the 128-bit registers and, in case more registers are needed than available, spilled to memory.

3 Optimizing Compiler for Vertex Shader Programs

This section describes the optimizing compiler. To experiment with new instructions, the code generator and instruction selector must be retargetable. The register allocator is also parameterized. The chosen optimizations and their implementation are guided by the retargetability requirements, and influenced by the data organization. A novel mask optimization is dictated by the unique semantics of mask and 'swizzle' operators in vertex shader instructions.

3.1 Internal Data Organization

There are two prevalent ways of arranging the several components of the vertex data for 3D algorithms like transformations and lighting: the array of structures (AOS), and the structure of arrays (SOA).

In the AOS organization each vertex is represented as a structure with several components (namely x, y, z and w). This corresponds to the intuitive notion in which each vertex is considered independently. The 3D object is represented as an array of such structures with dimension N corresponding to the number of vertices represented. Note that the AOS organization does not lend itself easily for SIMD processing as it is not always possible to apply the same SIMD operations to all components of a vertex. Furthermore, not all components of a vertex are operated upon in such a way that matches the (hardware-constrained) dimension of the SIMD instruction. This leads to computational inefficiency. For example, consider using an SSE instruction to scale the x, y, and z components of a vertex. The fourth multiplication result for the w component is wasted because it is never used.

The SOA organization uses an array for each coordinate. The whole object is represented by a structure composed of the component arrays, one for each coordinate. The dimension of each of the arrays is the number of vertices in the 3D object. Operations on a set of vertices are easily performed with SIMD operations on each of the component arrays. In the scaling example above, one would use three SIMD multiplications (one each for the x, y and z components) to create scaling results for a vector of vertices at a time. In this case the multiplication results are fully utilized.

Our compiler uses the AOS organization during code generation. Although the SOA organization in the compiler has some advantages, the AOS approach is the more promising approach for several reasons. First, due to its intuitive appeal, AOS is the prevalent programmer-defined data structure (for example, it is the data layout expected by the DirectX API). To use SOA, the graphics algorithms have to perform swizzle operations on the input data to convert from AOS to SOA for efficient processing using SIMD instructions. Consequently, the SOA approach incurs additional overhead due to the swizzle operations. Second, SOA processes four vertices at a time. This increases the register pressure and results in more register spilling during register allocation. Finally, the SOA organization works well when all of the vertices being processed follow the same control path through the programs. This was true in today's shaders because the vertex shader virtual machines do not support data-dependent branches. This means that the branches are determined by the constants loaded in the program and are independent of the vertices themselves. However, the trend in programmable shaders is towards increased flexibility and future shader virtual machines are expected to support data dependent branches.

3.2 Compiler Organization

The compiler is organized in three stages: the parser processes vertex shader programs and builds an intermediate representation, the optimizer transforms the intermediate representation and allocates registers, and then the code generator selects instructions and generates assembly output.

The *parser* leverages the DirectX reference rasterizer [4] which is basically an interpreter for vertex shader programs. As it examines each instruction, the interpreter is modified to build the intermediate data structures to be used by the optimizer. The reference rasterizer was modified so that it performs one pass over the whole program. This approach was selected to speed up development. To ensure correctness of the generated programs, their output is verified against the reference rasterizer.

To experiment with new instructions a retargetable *code generator* is utilized. It uses BURS technology and was built using the Princeton Olive [18] pattern-matching code generator-generator. In this method, machine instructions are described as a pattern to be matched in the intermediate data structure. The Olive input is a set of reduction rules for the instructions and the associated costs. When the code

generator find a minimum cost covering of the intermediate representation, the sequence of associated instructions corresponds to the generated code. New instructions are easily added by a new pattern the Olive input file.

Register allocation is done by a graph-coloring [1] register allocator along with several register coalescing passes. Spill code is generated when there are not enough physical registers.

A simple list scheduler reorders instructions in data flow order using a simple critical-path priority function. An aggressive list scheduler can greatly increase the register pressure and cause excessive register spilling. This is a problem since the ISA currently supports only 8 SIMD registers. To alleviate this problem, the list scheduler is designed to keep the register pressure small while scheduling instructions to reduce the execution time [5].

Other optimizations were considered. Software pipelining [9] overlaps execution of multiple loop iterations. For vertex shaders, each vertex is independent of the next. In this case, the compiler achieves inter-iteration overlap effects by unrolling loops and thus software pipelining was not implemented. The virtual machine has limited support for procedure calls, which may be nested to a limited depth. The compiler inlines all procedure calls.

Our experience with the optimizer is discussed in the subsequent sections. For comparison, an alternative naive implementation of the compiler used a 'macro expansion' approach. To convert vertex shader programs to 'C' code, each vertex shader instruction is translated into a sequence of high level language constructs, using compiler directives to specify SIMD operations. The output is then processed by an optimizing C compiler. In a rough comparison, the optimizer described in this section generates code that is about 70% faster than code using the macro expansion approach.

4 Architectural and Compiler Enhancements

To explore ways of improving the performance of the shaders, we investigated a number of compiler and architectural enhancements. This section describes three enhancements that were the most promising. Section 6 presents the performance impact of each of these enhancements.

4.1 New instructions

Dot product operations are very common in vertex shaders. A simple four-element dot product instruction in the vertex shader usually translates into about six instructions: one multiplication, three shuffles and two additions. The multiplication computes four element-wise products; two shuffles and two additions compute the intra-vector reduction and a final shuffle replicates the result in all vector positions.[4] Consider the following shader instruction that performs a dot product of `v0` and `v1` and stores the result in each of the four words of `oPos`:

```
dp4    oPos, v0, v1
```

This would create the following instruction sequence that computes the dot product of values in register `%xmm0` and `%xmm1` and stores the result in register `%xmm2`. It uses register `%xmm5` and `%xmm6` to hold intermediate values.[5]

```
movaps   %xmm5, %xmm0
mulps    %xmm5, %xmm1
pshufd   %xmm6, %xmm5, $14
addps    %xmm6, %xmm5
pshufd   %xmm5, %xmm6, $1
addps    %xmm5, %xmm6
pshufd   %xmm2, %xmm5, $0
```

Examining the generated code in detail revealed that dot product operations account for a significant fraction of instructions in the generated code. Consequently, we propose adding native support for dot product in the processor by supporting two new SIMD instructions: `dp3ps` and `dp4ps`. They perform dot product on the two source registers and store the result in each of the four words in the destination register. They differ in that `dp3ps` only uses three of the words in the source registers (vector of length three) while `dp4ps` uses all four words.

4.2 Mask Analysis Optimization

A number of generated instructions are extraneous because the compiler uses liveness information on a per register basis. Often, only part of a SIMD register is live. Keeping track of liveness information at a finer granularity (for each 32 bit word in the SIMD register) allows more effective optimization. Consider the following shader instruction:

```
mul    r0.xz, v0.xyz, v1.w
```

The semantics of the shader virtual machine specifies that this instruction is the same as the following instruction. It multiplies two vectors (`[v0.x,v0.y,v0.z,v0.z]`, and `[v1.w,v1.w,v1.w,v1.w]`) and stores the `x` and `z` components of the result into `r0`. It leaves the old values the `y` and `w` components of `r0` unchanged.

```
mul    r0.x_z_, v0.xyzz, v1.wwww
```

[4] The general case can be more complex because each of the inputs can have shuffle suffixes and the output can have a mask (Section 4.2).

[5] Brief notes to help follow the assembly code: (A) `xmmN` are SIMD registers. (B) `ps` suffix on instructions indicates a SIMD operation. (C) `pshufd` shuffles the contents of the source register and stores into the destination register. The shuffle is specified using a constant byte mask.

The compiler translates this shader instruction into the following instruction sequence[5]. The code assumes that at the `v0` is stored in `xmm0`, `v1` is stored in `xmm1`, and `r0` is stored in `xmm2`. It uses `xmm5` and `xmm6` to store intermediate values.

```
pshufd    %xmm5, %xmm0, $164
pshufd    %xmm6, %xmm1, $255
mulps     %xmm6,  %xmm5
andps     %xmm6, _Mask+80
andps     %xmm2, _Mask+160
orps      %xmm2, %xmm6
```

This example illustrates several sources of inefficiencies in the generated code.

1. The shuffle in the first instruction is not necessary. Note that the above shader instruction is equivalent to the following because the `w` component in the result is ignored.

   ```
   mul    r0.x_z_, v0.xyzw, v1.wwww
   ```

   Therefore, the first instruction in the generated code can be replaced by a move instruction (which will likely be eliminated during register coalescing).

   ```
   movaps    %xmm5, %xmm0
   ```

2. The second shuffle instruction can be eliminated if an appropriate vector was available because it was computed in an earlier instruction. The fact that the vector needed is `v1.w_w_` increases the likelihood of finding such a vector because the values in the `y` and `w` components of this vector does not affect the result.
3. The instructions to perform masking (`andps` and `orps`) might not be required. This happens if the `y` and `w` components in `r0` do not hold useful values either because they are never used or because they are overwritten by a subsequent instruction.

To eliminate the above inefficiencies due to mask and swizzle operations, a novel mask-analysis optimization was developed. This is a SIMD-specific optimization that exploits the fact that the different vertex components (x, y, z, or w) have useful values at different points in the program.

Mask analysis optimization involves two phases:

Analysis Phase. This employs a variant of live-variable analysis that keeps track of liveness information at a finer granularity— keep track of liveness information for each of the four 32-bit words in SIMD variables. It determines operations that produce component values that are not required by later operations. For each instruction, the liveness information is propagated from the destination to the source registers. For instructions that store to memory, all four components of the register value being stored are deemed to be live. For most instructions, the mapping of the liveness information is straightforward. For instance, in the following instruction, if the `x` and `z` components of `xmm6` register are live after the instruction, then the `x` and `z` components of registers `xmm5` and `xmm6` will be live prior to the instruction.

```
mulps     %xmm6,  %xmm5
```

For instructions that shuffle the register value, the mask constant is used to determine the liveness mapping. For instance, the following instruction takes the `w` component of register `xmm1` and stores it in each of the four words of register `xmm6`. So if any of the four words of register `xmm6` are live after this instruction, then only the `w` component of register `xmm1` will be live prior to the instruction. Otherwise, all the components of register `xmm1` will be dead prior to the instruction.

```
pshufd    %xmm6, %xmm1, $255
```

Optimization Phase. The liveness information computed on a per-word basis is used to eliminate the inefficiencies described earlier. Typically, the information allows shuffle and mask instructions to be replaced by register-to-register move operations that can be optimized away during register allocation. It also identifies dead instructions that can be immediately discarded.

4.3 Increasing number of SIMD registers

The generated code exhibits significant register pressure and causes register spilling during register allocation. Notice that the memory accesses generated by spilling registers to memory have a strong effect in performance. This effect becomes even more pronounced as CPU speeds become higher in relation to memory latencies. This suggests increasing the number of architectural SIMD registers provided from 8 to 16.[6]

5 Experimental Setup

Vertex shaders have to operate on every vertex of the scene that comes down the graphics pipeline. Consequently, their performance is crucial. Typical vertex shaders are small (less than 50 shader instructions). Often they do not contain any loops. Table 1 shows the list of 10 shaders used in this study.

The vertex shader compiler described in Section 3 has a number of knobs to control the optimizations implemented (like unrolling and list scheduling). The *base configuration* was selected by varying the various compiler parameters and selecting the one that yielded the best performance.

[6]The reason we do not consider increasing the number of SIMD registers beyond 16 is because the current ISA cannot be easily extended to support more than 16 SIMD registers.

Shader	Number of Shader Instructions	Loops?
Blur	9	No
CookTorCube	84	Yes
Lighting201	65	Yes
PaletteSkin	49	No
PointLight2	24	Yes
PredatorEffect	73	No
Ripple	34	No
Tetrahedron	28	No
ToonShade	20	No
World	8	No

Table 1. Vertex Shaders

All numbers reported in this paper use the same parameters for the compiler as in the base configuration (excluding the parameters that control the enhancements discussed in Section 4).

The compiler translates the vertex shader into an assembly function that implements the shader's functionality. To measure its performance, it was compiled together with a main program that repeatedly invokes it with input vertices. The main program has a loop operating on an array of vertex input data and producing as a result an array of vertex output data. The input vertices are initialized to random values. Recall that the flow of control in the vertex shader is independent of the input vertex, so the actual values have little impact on the execution.[7]

The memory access pattern of the vertex shader tends to be quite predictable. A stream of vertices is processed one vertex at a time while accessing only a limited amount of memory. This makes it easy to use prefetching techniques so that the memory accesses are satisfied entirely by the caches. To simulate this, the array of vertices is kept small enough to fit in the cache. An initial pass of the array of vertices is performed to ensure that all memory locations are cached in memory while measuring the performance of the vertex shaders.

The performance of the vertex shaders was measured on a 2.2GHz Intel Pentium® 4 system which supports the SSE2 instruction set.

6 Performance Evaluation

This section presents the performance of the vertex shader in the base configuration. It also discusses the performance impact of the three enhancements discussed in Section 4.

[7]Some care is taken in limiting the range of the random numbers to avoid getting an out of range index for accessing the constant memory or a division by zero.

Base Configuration Table 2 shows the performance of code generated by the compiler in the base configuration (Section 5). The performance varies between 25 (for Blur) to 0.3 million vertices per second.

Table 2 also shows the breakdown of instructions in the generated code. The SIMD instructions account for a large majority (83-99%) of instructions in the generated code. Two other points are worth noting. First, on average, each shader instruction translates into around 10 instructions in the generated code. This expansion is due to instructions to mask and combine vector fields for the mask operators, intra-vector element shuffles to handle swizzles, and occasional register spills to memory. Second, the generated code has very large basic blocks, as most shaders do not have control flow. Even the three shaders that have control flow have only a few control instructions.

The rest of this section evaluates the performance improvement that can be achieved over the base case by enhancing the architecture and the compiler. Figure 4 shows the performance impact of two of the enhancements on the performance of the shaders. Figure 5 shows the impact of the three enhancements on the number of instructions in the generated assembly code.

New Instructions. Including these two instructions (`dp3ps` and `dp4ps`) reduces the number of assembly instructions by 23 % on average (Comparing A & B in Figure 5). Since a dot product instruction replaces a combination of "SIMD Arithmetic" and "SIMD Shuffle" instructions, they account for most of the reduction in the number of instructions.

The performance impact of adding these two instructions (`dp3ps` and `dp4ps`) to the instruction set cannot be directly measured on current processors because they are not supported by current processors. However, instruction `mulps` (it multiplies two SIMD values) in the current instruction set provides a reasonable approximation to the behavior of the `dp3ps` and `dp4ps` instructions. The latency of the proposed instructions is expected to be similar to a multiplication, and the set of argument and result registers is similar.

Although the resulting value would be wrong, it does not have affect the program behavior because the control flow in today's shaders cannot be influenced by values generated in the program (Section 2). The performance is measured by replacing `dp3ps` and `dp4ps` instructions by `mulps` instruction in the generated program and running it. Figure 4 shows that adding these two instructions would improve the performance of the shaders by 1-51 % (27.4 % on average).

Mask Analysis Optimization. Performing mask analysis

Shader	Number of Assembly Instructions							Processing Time Per Vertex (in μs)
	Total	SIMD				Scalar		
		Arithmetic	Shuffle	Memory	Spills	Control	Rest	
Blur	160	34	54	44	0	0	28	0.04
CookTorCube	1973	345	1211	51	310	4	52	3.37
Lighting201	2918	545	1686	142	258	12	275	3.14
PaletteSkin	1133	230	489	112	190	0	112	0.40
PointLight2	475	122	212	39	26	9	67	0.52
PredatorEffect	1266	266	664	92	172	0	72	0.49
Ripple	516	84	333	32	59	0	8	0.20
Tetrahedron	597	102	306	62	119	0	8	0.22
ToonShade	464	100	231	48	77	0	8	0.16
World	216	34	101	32	41	0	8	0.06

Table 2. Generated code in the base case. The instruction categories are explained in Table 3.

Instruction Category		Description
SIMD	Arithmetic	SIMD arithmetic instructions including those that perform add, multiply, and square root.
	New	Two new SIMD instructions proposed in Section 4.
	Shuffle	SIMD instructions that select individual words from the source registers, reorder them, and store them in the destination register. Includes simple register to register move instructions.
	Memory	Instructions that load and store SIMD values to Memory. Memory operations due to register spills are not included.
	Spills	Instructions that load and store SIMD values to memory due to register spilling.
Scalar	Control	Jump instructions
	Rest	Remaining scalar instructions

Table 3. Description of the instruction categories used in Table 2 and Figure 5

improves the performance of the shaders by 19.5 % on average. It reduces the number of instructions in the generated code by 21 % on average (Comparing A & C in Figure 5). As expected (Section 4), the bulk of the instructions eliminated belong to the "SIMD Shuffle" category.

Increasing number of SIMD registers. Increasing the number of SIMD registers from 8 to 16 reduces the number of memory accesses due to register spills by 35-100 % (Comparing A & D in Figure 5). Although the percentage of instructions due to spills is small, the spills are significant because they involve memory accesses.

We cannot measure the impact of increasing the number of SIMD registers from 8 to 16 on a real machine. Therefore, we have modified a microarchitecturally correct simulator that simulates a Pentium®-like machine to support 16 SIMD registers. Our preliminary measurements indicate that the impact of reducing the number of register spills is quite significant. For instance, for the *PointLight2* shader, the execution time for processing each vertex reduces by 11 %. In this shader, the number of spill instructions (as a percentage of the total number of instructions in the generated code) reduced from 5.47 % to 0 %.

Increasing the number of SIMD registers will have a bigger impact when instruction scheduling is turned on. Instruction scheduling often increases the register pressure resulting in an increase in the number of registers spilled in the program. With 8 SIMD registers, the instruction scheduler resulted in an overall performance degradation on the shaders. Consequently, the base configuration has the instruction scheduler turned off.

Overall. Performing the first two enhancements (new instructions and mask analysis) together yields a performance improvement of 42 % on average. Performing all three enhancements results in a reduction in the number of instructions by 46.7 % on average. Note that the three enhancements are fairly orthogonal and target different categories of instructions.

Another interesting point that is evident from Fig-

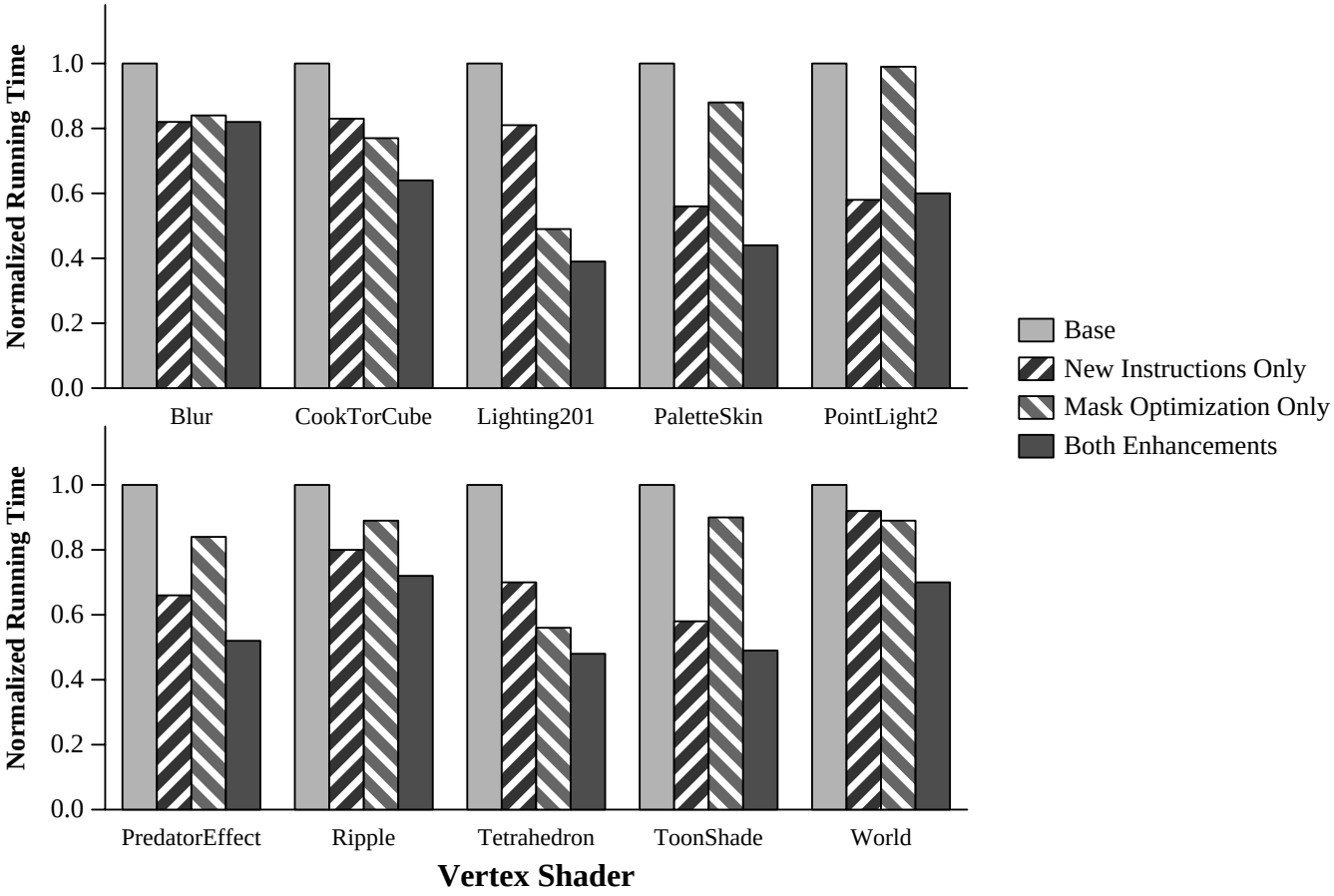

Figure 4. Performance improvement achieved for the different vertex shaders due to two architectural/compiler enhancements.

ure 5 is that the "Shuffle" instructions account for a large percentage of instructions even after the various optimizations. SIMD instructions deliver better performance by operating on more than one (4 on Pentium®) data units at a time. However, this benefit can be eroded if it requires a lot of "shuffle" instructions to compose the data into and out of the SIMD registers to enable the SIMD operations.[8]

7 Related Work

The goal of this work is to obtain efficient execution of graphics programs on an existing general-purpose CPU. The approach is to identify incremental enhancements to the architecture along with associated compilation techniques.

This relates to and is in contrast with (micro)architectural specialization for a given application domain such as the application-specific specialization of [13] or the tailored instruction sets of [7]. Domain-specific processor specialization is also exemplified by the prevalence of architectures such as DSPs and vector processors.

Previous approaches to efficient program generation for graphics architectures range from language design issues [6] to compilation techniques [11]. In contrast, our focus is on taking advantage of application characteristics to extend compilation techniques, as exemplified by our mask optimization algorithm.

The vertex shader language used as input is unusual because it expresses a degree of implicit parallelism in the form of assembly-like SIMD operators. The compilation task is one of matching the general purpose SIMD instruction set to the SIMD operations implied by the language applications. In this case the programmer expresses paral-

[8]Not every shuffle instruction in the program impacts its performance. This is because the SIMD instructions are long latency instructions and occupy the arithmetic units for a number of cycles. This means that two consecutive SIMD arithmetic instructions cannot be issued in two consecutive cycles. The shuffle instructions can be executed in the intervening cycles without impacting total execution time.

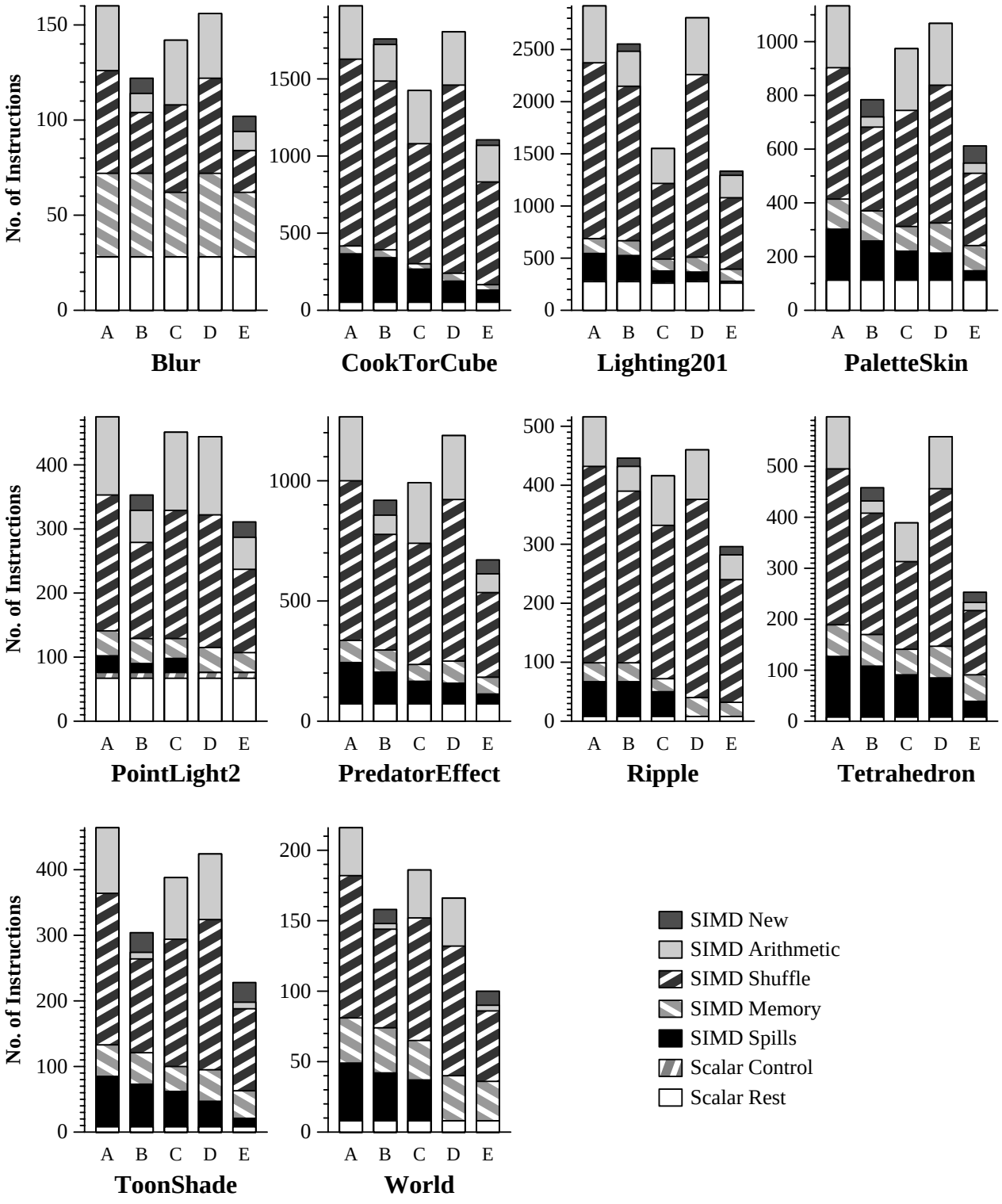

Figure 5. Breakdown of the instructions in the generate assembly code in five configurations: (A) Base (B) New Instructions Only (C) Mask Optimization Only (D) 16 Registers Only (E) All three enhancements. The instruction categories are explained in Table 3.

lelism and the compiler may treat the optimization task as one of instruction selection and code generation, instead focusing on extracting low-level ILP.

8 Conclusions

Graphics and Media processing is a workload of increasing importance for general-purpose processors. This paper focuses on architecture and compiler enhancements to enable effective support for graphics programs on a general purpose CPU. We present the execution characteristics for a number of representative vertex shaders.

Our results show the importance of support for dot product operations and increased register file size. A novel and effective compiler optimization is also described. This paper discusses the impact of these three changes in the architecture and compiler. The measurements indicate a potential two-fold speedup due to these changes. Adding support for new specialized instructions improves the performance of the programs by 27.4 %. on average. A novel compiler optimization called mask analysis improves the performance of the programs by 19.5 % on average. Increasing the number of architectural SIMD registers from 8 to 16 registers significantly reduces the number of memory accesses due to register spills.

Acknowledgments

We would like to thank Patrice Roussel, Tom Huff, Dean Macri, Ronen Zohar, Ram Nalla, and Gordon Stoll for the many discussions and their feedback on the research presented in this paper.

References

[1] G. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1982.

[2] J. H. Clark. The geometry engine: A vlsi geometry system for graphics. In *SIGGRAPH*, 1982.

[3] K. Diefendorff and P. K. Dubey. How multimedia workloads will change processor design. In *IEEE Micro*, 1997.

[4] R. Fosner. Directx 6.0 goes ballistic with multiple new features and much faster code. In *Microsoft Systems Journal*, Jan 1999.

[5] J. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ACM SuperComputing*, 1988.

[6] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *SIGGRAPH*, 1990.

[7] B. K. Holmer. Automatic design of computer instruction sets. In *MICRO 27*, 1994.

[8] IdSoftware. Doom iii. In *www.idsoftware.com*, 2002.

[9] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[10] E. Lindholm, M. J. Kligard, and H. Moreton. A user-programmable vertex engine. In *International Conference on Computer Graphics and Interactive Techniques*, 2001.

[11] W. R. Mark and K. Proudfoot. Compiling to a vliw fragment pipeline. 2001.

[12] Microsoft. http://www.microsoft.com/windows/directx/. 2002.

[13] N. Moreano, G. Araujo, Z. Huang, and S. Malik. Datapath merging and interconnection sharing for reconfigurable architectures. In *Proc. of the 15th. ACM/IEEE International Symposium on System Synthesis*, 2002.

[14] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. In *IEEE Micro*, June 1996.

[15] M. Phillip, K. Diefendorff, P. Dubey, R. Hochsprung, B. Olsson, and H. Scales. Altivec technology: Accelerating media processing across the spectrum. In *HOTCHIPS X*, Aug 1998.

[16] P. Ranganathan, S. Advi, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media isa extensions. In *ISCA*, Sep 1999.

[17] S. Thakkar and T. Huff. The internet streaming simd extensions. In *Intel Technology Journal*, 2nd quarter 1999.

[18] S. Tjiang. http://www.ee.princeton.edu/spam/. 2002.

[19] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He. Vis speeds new media processing. In *IEEE Micro*, Aug 1996.

[20] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum 20*, 2001.

[21] R. Zohar. Personal communication. 2003.